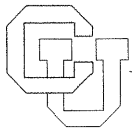
Geometry-Specific Languages and Their Interfaces

Michael Eisenberg
Tom Wrensch
Glenn Blauvelt

CU-CS-886-99



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

Geometry-Specific Languages and Their Interfaces

Michael Eisenberg, Tom Wrensch, and Glenn Blauvelt
Department of Computer Science and Institute of Cognitive Science
Campus Box 430
University of Colorado, Boulder CO USA 80309-0430
duck;wrensch;zathras@cs.colorado.edu; 303-492-1218
Technical Report CU-CS-886-99

ABSTRACT
Traditionally, programming languages have been designed with an eye toward implementation on general-purpose computers. The advent of *computationally-enhanced craft items*–small programmable objects with simple geometries and scaled-down computational components–suggests the need for new language and software environment models tailored for these objects. In some respects, the task of the craft-item language designer is less demanding than that of the traditional language designer; but there are new challenges to be faced as well. In this paper, we summarize both our experiences in creating computationally-enhanced craft items and their ramifications for the creation of geometry-specific languages, programming environments, and interfaces.

Keywords
Computationally-enhanced craft items, geometry-specific languages, computation and crafts.

INTRODUCTION
Programming languages have traditionally been a central study in computer science–indeed, the design of notations in which to express algorithmic ideas is a concern that dates back to the classic 1842 paper by Countess Lovelace describing the (proposed) architecture of Charles Babbage's Analytical Engine [12]. Courses in comparative programming languages often begin here (or with the work of Zuse in the 1940s) and proceed through an examination of early high-level languages (FORTRAN, Lisp, Algol 60), followed by modern representatives of the major language "families": imperative languages (Pascal), object-oriented languages (C++, Java), functional languages (Scheme, ML), logic languages (Prolog), visual languages, and so forth (cf. [10]).

While such comparative courses often emphasize the astonishing variety of the language landscape, there are a number of central assumptions behind virtually all the examples; some of these assumptions are so deep-rooted that they are rarely made explicit. Among these assumptions are:

- High-level programming languages are designed for use on *general-purpose* computers; indeed, this is the historical reason for their development. A language such as Pascal or Lisp may be implemented in virtually any hardware device capable of the operations of an abstract Turing machine.

- In those cases (e.g., visual languages) in which a language design assumes more than the most abstract computer architecture, it is still unlikely to assume anything other than the capabilities of standard desktop devices.

- Programming languages are designed to address recurring problems in the practice of professional software engineeering: how to write large and complex programs; how to reuse reliable chunks of existing code; how to avoid, discover, trace, and repair program bugs; and how to facilitate communication among programming teams. In response to problems of this kind, programming language designers have introduced notions such as static typing and type inference; block structure; classes, methods, and inheritance; modules and packages; and exception handling. Concerns of this sort surface even in the design of educational or end-user languages such as Logo, spreadsheet languages, and Visual Basic.

- The interfaces associated with software development environments are likewise tailored toward the concerns of the professional software developer: version management systems, steppers, debuggers, and design-specificiation systems are all created with an eye toward fragile, large-scale, complex programs on general-purpose devices.

Over the past two years, we have been compelled to reexamine these assumptions as a consequence of our

efforts to design *computationally-enhanced craft items* (CECIs). The idea behind these objects is that they are programmable versions of the sorts of small, inexpensive items typical of craft work and homespun design–tacks, hinges, tiles, and so forth. In the course of creating and writing sample programs for our prototype CECIs, we have observed that the classical tenets of both programming language and programming interface design are only imperfectly applicable to the task of devising notations for these objects.

Our view is that CECIs, in one form or another, are likely to become an increasingly prevalent presence in the worlds of home crafting and end-user programming. As such, it is perhaps not too early to explore, proactively, the problems involved in designing languages, environments, and interfaces for these devices. This paper is a discussion of what appear to us to be the important or recurring issues in this task; though we have to confess that our own ideas are still tentative, and based on the particular objects that we have designed.

The following (second) section of this paper describes in more detail the notion of a computationally-enhanced craft item and describes its relation to broader notions of ubiquitous and embedded computation; we also summarize our own efforts in CECI design. The third and fourth sections discuss in turn the implications of CECIs for the design of programming languages and programming interfaces and environments, focusing particularly on those aspects in which the conventional wisdom is likely to need reevaluation. Finally, in the fifth section, we describe several (in our view) major short- and medium-term challenges that underlie the creation of viable "geometry-specific languages" suitable for use with CECIs.

COMPUTATIONALLY-ENHANCED CRAFT ITEMS

The intent behind CECIs is to exploit the increasingly protean architecture and appearance of computers–the ability to endow objects of many sizes, shapes, materials, and uses with (at least small amounts of) computational capabilities. The impulse toward integrating computers into a variety of physical objects is likewise seen in the research of those interested in ubiquitous computing [16], microelectronicmechanical systems (MEMS) and "smart materials" [1], and augmented reality [9], among others. While the notion of CECIs shares some aspects of these other areas, it diverges in some important respects as well:

(1) CECIs are, in general, not intended to perform complex computations; that is, they are not conceived as powerful general-purpose computers shrunk to tiny size, performing large-scale tasks in planning or search. Rather, they are intended to perform simple, visible, and understandable operations or movements associated with common or everyday items such as hinges, strings, and tacks. Thus, instead of embedding a high level of intelligence in small objects (the notion behind many computational toys and appliances, for example), the CECI designer does not wish to make objects appear complex or "magical", but rather to present a deliberately accessible and "mechanical" user model of the object.

(2) CECIs are intended to be end-user-programmable devices, suited to the home scientist, craftsperson, hobbyist, or student. In this respect they differ from many of the devices seen in ubiquitous computing or augmented reality research, whose constructions are often aimed at either audiences uninterested in programming or (alternatively) at small, highly-trained professional communities.

(3) The computational capabilities of CECIs, while simple, are neither molecular in scale nor highly distributed (both of which are typical of work in MEMS). Conceptually, a CECI is a single discrete object with a simple, localized program.

Two examples may help to concretize this description. The programmable hinge [18] is a prototype device (in the shape of a hinge) that allows the user to program its opening and closing behavior. Our current version of the hinge employs a small computer–one of the "cricket" Lego bricks developed by Resnick and his colleagues at the MIT Media Lab[14]–and two strips of shape memory alloy that (alternately) open and close the hinge when heated by an electric current. (A schematic and photograph are shown in Figures 1 and 2.) The rototack [17] is a tack-shaped device that allows the user to program rotational behavior around its central axis. Our current prototype contains a PIC16LF84 chip (by Microchip, Inc.) that holds the computer program and a stepper motor that turns the head of the tack. (A schematic and photograph are shown in Figures 3 and 4.)

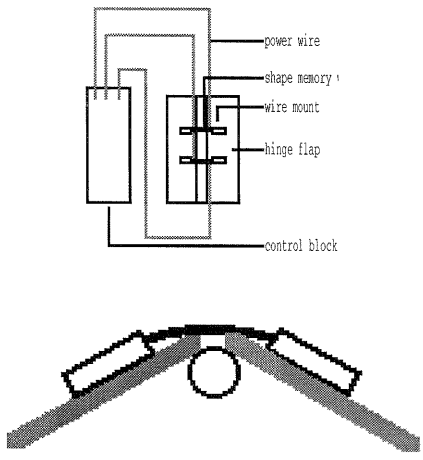


Figure 1. A schematic diagram of the programmable hinge (top), and (bottom) a sketch of the hinge as viewed from above.

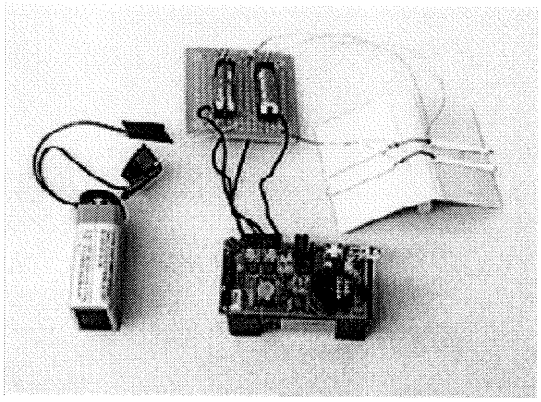


Figure 2. A photograph of the programmable hinge prototype. The hinge's battery power source is visible toward the left of the photo; the associated computer (a Media Lab cricket) storing the hinge's program is at bottom center; and the hinge itself, with shape memory wire actuator, is seen toward the right.

A typical program for the tack–typical in length, complexity, and behavior–might appear as follows:

```
A := 0
Loop
 Repeat 3
   Clockwise
   Turn 90 degrees
   Wait 10 tenths
 End Repeat

 Repeat 2
  Counterclockwise
  Turn A degrees
  A := A + 45
  If  A = 360
      Then A := 0
      End If
  Wait 20 tenths
 End Repeat
End Loop
```

In prose, this program loops indefinitely with the following pattern: the tack turns clockwise three times by 90 degrees, pausing briefly after each turn; then the tack turns counterclockwise twice by a variable amount (the amount increases by 45 degrees at each turn until reaching 360, at which point it returns to 0), pausing a bit longer after each of these two turns. A program for the hinge might be similar in structure (our current prototype is in fact programmed in Logo), but employing actions of "Open" and "Close" instead of turning.

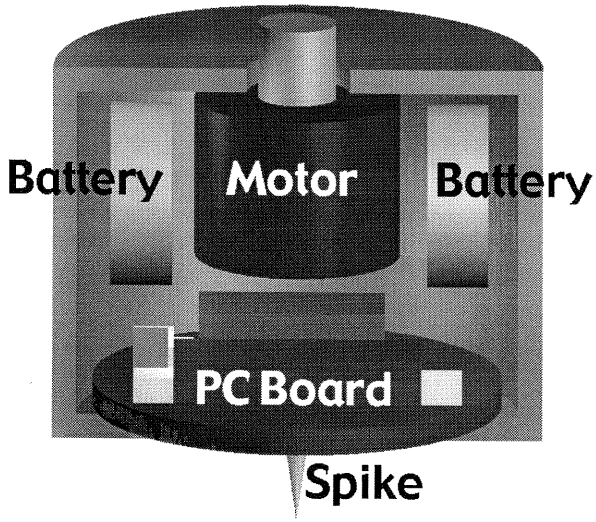


Figure 3. A schematic cross-section diagram of the rototack, showing the PC Board containing the PIC microcontroller, battery, and stepper motor.

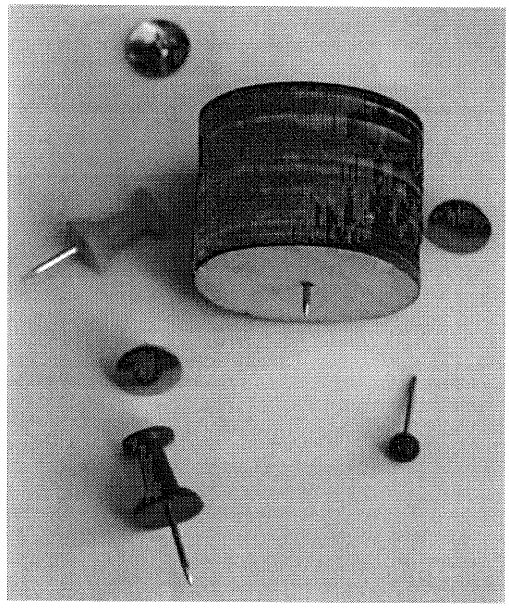


Figure 4. A photograph of the rototack (shown with a variety of other tacks for comparison).

While these two examples do not begin to exhaust the design space of CECI items (we have also, for example, built a simple prototype programmable ceramic tile that flashes an LED light under computer control), they should serve to motivate the discussion of programming languages in the following sections.

PROGRAMMING LANGUAGES FOR CECI'S: CONSTRAINTS AND PRINCIPLES FOR DESIGN

The previous section outlined the central ideas behind the construction of CECI's. Our current programming notations for the existing prototype objects are ad hoc artifacts, still in development: the hinge is programmed in a dialect of Logo, the tack in an imperative (somewhat Basic-like) language compiled into a byte code for which we have written an interpreter for the PIC microcontroller. Nonetheless, it is possible to step back from our experience thus far and describe at least some guiding principles for designers of CECI-appropriate programming languages.

(1) CECI languages do not fall neatly into either of the traditional categories of "high-" or "low-level". In some respects, these languages are high-level: not only should they employ many of the staple control constructs of traditional languages (Repeat, While, Loop, If/Then), but they also should include primitive procedure names for what might well refer in practice to rather complex real-world physical motions ("Open" for a hinge, "Turn" for a tack, and so forth). On the other hand, CECI languages have some of the flavor of low-level assembly code notation: depending on circumstances, the user might well be limited to integer-valued numbers (e.g., one could not program the tack to rotate 22.5 degrees) or she might be expected to limit the number of variables (such as A in the earlier example) to a very small set. More generally, CECI languages are "low-level" in the sense that they are intimately associated with a particular piece of hardware: a particular geometry, range of behavior, and (perhaps) computational architecture, as we discuss in the next point below.

(2) Because CECI languages are developed to describe the (highly limited) behaviors of a simple craft object, they are crucially constrained by the geometry of that object. For this reason we have used the term *geometry-specific language* to indicate the close relationship that will necessarily exist between these notations and the objects for which they are built. We expect that the number of interestingly distinct geometry-specific languages will be relatively small (perhaps 20 or 30). A language for spherical objects such as ball-bearings, for instance, might be based upon rolling behavior; a language for string might be based upon discrete "tugs"; a language for rods or pistons upon "push/retract" motions in one dimension; and so forth. Quite possibly, a taxonomic structure of the kind used by Card, Mackinlay, and Robertson for I/O devices [5] could be profitably devised for mapping out the space of craft-item geometries and behaviors.

(3) CECI programs are generally realized in visible, overt behavior of physical objects. The sample program of the previous section is illustrative: statements such as "Turn 90 degrees" and even "Wait 10 tenths" correspond to observable physical behaviors. Such programs thus spend little or no time "just thinking" in processes such as search or in lengthy numerical computations. This in turn has ramifications for the design of CECI language interfaces, as we will discuss shortly.

(4) The computational resources of CECI languages are assumed to be minimal; indeed, the language designer might, for a variety of reasons, enforce a user model that is less powerful than the actual hardware can accommodate. For instance, programs might be limited to a page in length; to simple data structures (such as one-byte integers); to a small number of user-defined procedure names; to a low degree of nesting of control structures; and so on. The upshot of these limitations is that many of the features of traditional languages whose purpose is to manage program size and complexity are no longer required. A language that only employs small integers as variables needn't include type declarations or polymorphic functions; a language whose programs are all at most a page in length needn't include packages or modules; a language in which user-defined names are few in number needn't worry about variable scoping, since it is feasible to require that every variable have a distinct name. These observations might of course be uncharitably construed as making a virtue out of grim necessity: if the language is expressively impoverished, so the argument goes, then of course it needn't include the encumbrances of more powerful languages–this is rather like saying that the Roman numeral system is "unencumbered" by the need to express the number zero! There is some truth to this response, but we would argue that the simplicity of CECI languages realistically reflects the uses to which programmable craft items are likely to be put. Moreover, the structural limitations of these languages affords interesting possibilities for growth in complexity and sophistication of their associated environments and interfaces, as we will argue in the succeeding sections.

IMPLICATIONS OF GEOMETRY-SPECIFIC LANGUAGES FOR ENVIRONMENT AND INTERFACE DESIGN

The thrust of the previous section's discussion was on the simplicity and expressive limitations of geometry-specific languages for craft items. In this section, we turn our attention from the features of the languages *per se* to those of the larger systems–the language environments–which the user of CECIs should interact with.

It is worth pausing at this juncture to highlight two central prefatory themes. First, while a CECI program is written for, and is run on, a simple craft item like a hinge or tack, the text of the program itself is written by the user on a desktop workstation, and subsequently transferred to the craft item. Thus, while the size and cost of a CECI may pose severe constraints for the *content* of a program, they pose no obvious constraint to the software environment in which the program is created by

the user. This situation is of course familiar to software professionals such as game programmers who write code on one machine to be run on another (usually simpler or more specialized) device. Still, it is a relatively new situation for end-user programmers (although the advent of objects such as the programmable Lego brick [14] does change the landscape somewhat).

Second, the formal distinction between "language" and "language environment", while undoubtedly useful for some purposes (e.g., presenting a compact language specification), is not always clear-cut. Much of the work in visual language design, for instance, is hard to pigeonhole as focusing either on "language" or "environment" in isolation. Moreover, as far as the user is concerned–particularly an end-user of something like a spreadsheet or application-specific language–the distinction may be close to academic: to such a user, the "language" is simply the entire programming system represented on the screen and in the manual. And since CECI languages are indeed conceived as end-user languages, it seems only reasonable to consider the design of the environment and interface in concert with the design of the language itself.

These two themes–the availability of computational resources for CECI language environments, and the tight interweaving of CECI languages and interfaces–underlie many of the points to follow.

(1) Traditional programming environments are designed to enforce an abstraction barrier between the user's view of the language and her view of the machine on which the program will be run. Thus a Java or Scheme programmer (and to a slightly lesser extent a C programmer) typically has no idea, and no need to know, anything about the machine architecture on which she is working: the clock speed, word size, number of stack registers, and so forth. While this might occasionally deprive the high-level language programmer of some ingenious machine-specific hack, it would be the rare (and dubious) program whose meaning or correctness depends on such knowledge. Geometry-specific CECI programming environments, in contrast, present a somewhat different problem to the programmer. A rototack programmer, for instance, might wish to take into account the specifics of the hardware on which her program will be run: some stepper motors (to pursue the example) might turn a tack by 5-degree increments, others by 10-degree increments. In the latter case, a program that specifies a 15-degree turn would behave differently than the programmer desired. A hinge programmer might wish to have some idea of the response time of the device: how long will an "Open-Hinge" command take to complete?

Again, such situations are hardly new to (e.g.) chip designers, for whom there is a close relationship between the constraints on programming and the specific architecture of the machine on which the program will be run. But these are relatively new concerns for end-user programmers. The situation is complicated a bit more by the likelihood that a CECI programmer might wish to send a program to several variations of a particular craft item: for example, one might have a few sizes or generations of rototacks on hand, and might wish to endow them all with the same general-purpose program. In such a case, the programming environment should have means of communicating to the user what hardware requirements are stipulated by a given program: e.g., in our earlier rototack program example (which involved turns of multiples of 45 degrees), the environment should indicate to the user that any variation of programmable tack on which this program is run needs to accommodate turns in multiples of (say) 1, 5, 15, or 45 degrees. In other situations, the CECI environment might usefully take into account other physical features of the craft item–its size, or weight, or material composition.

(2) Because CECI programs maintain a close correspondence with physical effects, it is useful for a CECI programming environment to juxtapose or integrate program code with a graphical representation of the running program. Figure 5 depicts how the screen interface of such an environment might look for a tack-programming system. Here, the program code is shown in one frame, while an abstract line-drawn representation of the tack is shown in another. For such a system, the user could elect to run his program in "animation" mode on the line-drawn figure before sending it to the physical craft item.

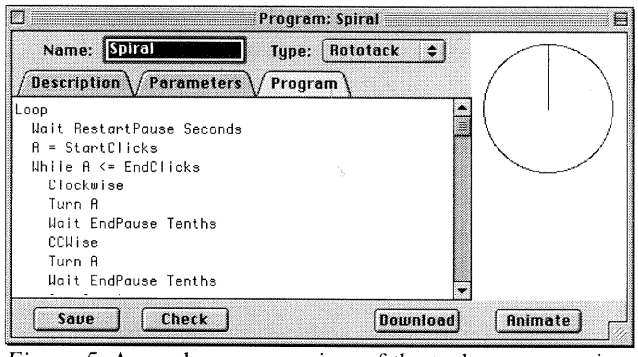


Figure 5. A mockup screen view of the tack programming environment described in the text. The frame at left shows the text of the program in construction (views of the program's parameters, and a textual description of the program are also available in this frame). At right, an abstract representation of the tack (as seen from the top). The Animate button toward the bottom right links the running program to the graphical tack.

In programming environments for traditional general-purpose languages, there have been similar efforts to juxtapose running programs and visual representations, sometimes for educational purposes. (See, for example, [3, 4].) A recurring difficulty in such systems–and for the task of algorithm animation more generally–is that the

environment designer must create visual representations for a variety of computational structures and concepts (lists, trees, arrays, memory locations, and flow of control, to name a few). In CECI environments, by contrast, the mapping between code and animation is–if not immediate–at least somewhat more straightforward. A program like the earlier rototack example need only correspond to animation of an abstract tack in the act of turning and pausing; more detail than this (e.g., the value of a variable such as A in the earlier program, or whether the tack is set to turn clockwise or counterclockwise) might be worthwhile to depict, but even an extremely simple interface without these details would likely be useful. This suggests that many of the techniques and insights developed in the algorithm animation research community (e.g., for linking multiple representations) might profitably be transferred to the less complex tasks of CECI-program animation.

(3) Pursuing the ideas of the previous paragraphs, we could exploit the simplicity and "animatability" of geometry-specific languages in still other ways. For instance, since CECI programs are largely distinguished by their physical effects–by how they make a simple object behave–it might be of interest to use animations as indices into a library or repository of CECI programs. A program browser for software reuse, then, would consist of a palette or scrollable list of animations (e.g., of a rotating tack). By choosing the animated motion closest to the type of effect that she is seeking, the user could then bring up and (if necessary) edit the program that gave rise to the chosen animation. Here, the task of software reuse is made simpler because CECI programs are short and concrete; hence, entire programs (rather than routines, algorithms, or objects) constitute the grain at which code is stored and retrieved. Along the same lines, the fact that there are likely to be only a moderate number of distinct program types suggests that one could experiment with qualitative summaries of CECI behaviors as indices to software reuse: for instance, the user might specify that she wishes to look at those programs in which a "tack rotates slowly back and forth" or in which a "hinge opens and closes at random time intervals".

(4) Yet another body of research that could be explored (or revisited) in the context of CECI languages involves the long-held desire to construct programs by example or demonstration (cf. the especially provocative set of papers edited by Cypher [6]). Nardi's [13] discussion of this area, and of the difficulties that designers of such systems face, is especially thoughtful: in particular, she cites the problems involved in having a system infer constructs such as conditionals from examples, and of resolving ambiguity and ignoring errors and slips in user input. While these problems are unlikely to disappear entirely in the context of CECI programming, they are arguably less daunting in that context. First, many CECI programs might well be simple enough so that they would not even require constructs beyond simple iteration: inferring a program merely to rotate a tack 270 degrees in three 90-degree steps requires far less sophistication than inferring a program like the sample shown earlier. Second (and in accord with Nardi's overall position), integrating textual and demonstrational programming might be achievable in a CECI language environment: moving an icon (like that of the tack) by direct manipulation could create code which could then be studied and edited, much as in the commercial AutoCAD program (as well as Lieberman's MONDRIAN graphical programming system [11] and DiGiano's Chart 'n' Art system for generating charts and information displays [7]).

(5) The previous paragraphs argue that geometry-specific languages offer especially appropriate venues for experimenting with (or revisiting) ideas in software library construction, algorithm animation, and programming by demonstration. On the other hand, there are several staples of sophisticated programming language environments that would, quite probably, be far less important in the context of CECIs, and whose excision would simplify the construction of a CECI language system. In particular, since the programs being created are so brief, since their associated data structures are so few and simple, and since their effects are so directly observable, the need for sophisticated steppers, debuggers, and editors would be much reduced. There would be far less need for ancillary tools such as those used for maintaining and checking specifications; and the interface to a CECI language compiler or interpreter would be streamlined as well (it is hard to see a need for special tools such as incremental compilation when programs are no more than a page in length). In brief, then, the designer of a full-featured CECI language environment is likely to face challenges that have generally been regarded as experimental, while dispensing with at least some other features regarded as mainstream (if state-of-the-art) additions to modern environments.

FUTURE CHALLENGES FOR THE GEOMETRY-SPECIFIC LANGUAGE DESIGNER

So far in this paper, we have focused our discussion on the shorter-term issues involved in creating geometry-specific languages and programming environments. In this final section, we outline several broader research areas suggested by the discussion thus far.

• *CAD Systems for Programmable Objects.* For each of the prototype CECIs that we have constructed–tack, hinge, and tile–we have had to devise a set of appropriate commands with which to program the object. Throughout this paper, the process of linking a programming language with a new craft item has been likewise presented as an ad hoc, one-device-at-a-time process: thus, if a designer were now to create (e.g.) an architecture for a programmable bolt, she would have to create a new language for that object as well. Clearly, as the number and variety of CECIs expands (and we hope that it will), there will be a growing redundancy of effort represented by

the construction of so many "little languages" with which to program them.

Conceivably, then, it would be useful to explore methods by which the process of creating a brand-new programmable object–with an associated geometry and set of programming commands–could be streamlined. One way of doing this might be to create a "programmable object CAD system" in which not only are new geometric objects created (as in standard CAD systems), but simple dynamics of those objects are associated with textual commands that are embedded in a larger language (such as a dialect of Logo, Basic, or Java). Figure 6 may help to elaborate on this idea: this is a screen shot of what such a "CECI-CAD" system might look like. In this scenario, the user has created a geometric representation of a bolt (via the types of constructions typical of CAD systems), but in addition specific dynamic behaviors (extending and retracting the bolt, locking and unlocking its current position) have been created and given names that could be inserted in a language much like our current rototack dialect.

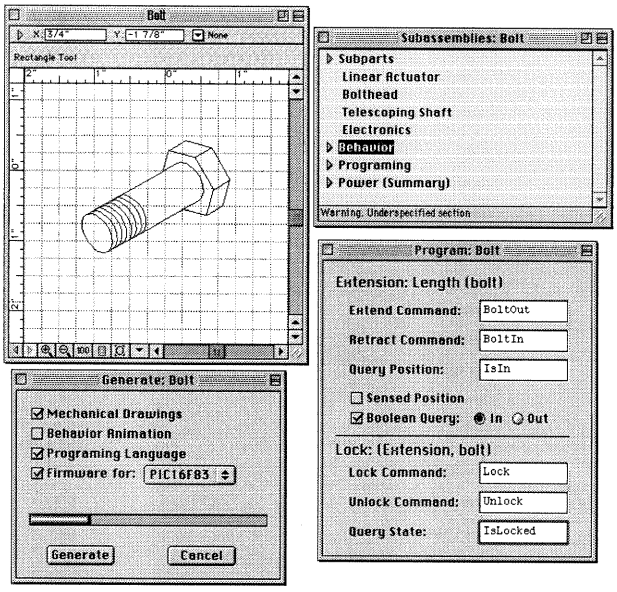


Figure 6. A mockup screen view of the CECI-CAD program described in the text. The user creates a geometric shape (such as the bolt at upper left) and endows geometric changes to that shape with procedural names in the window at bottom right. The intent would be to produce programmable objects with novel geometries along with their associated language environments.

A system of this sort could serve as the basis of a process by which new programmable objects could be designed: the result of a session at CECI-CAD would be a specification of a solid shape, its associated movements, and its associated language or dialect. While the specific hardware of the physical object itself–the actual stuff that is seen (e.g.) in Figures 2 and 4 earlier–would still have to be designed individually, the language environment with which to program the new object would not have to be created from a blank slate.

• *Craft Objects and Languages for Children.* Our discussion in this paper has portrayed CECIs primarily by contrast with the sorts of objects that one might find at a crafts or hardware store–objects aimed at an adult audience. Our initial entry into the subject began, however, through an interest in the design of applications that integrate software and educational crafting activities. There is a venerable tradition of marvelous crafts for children–mathematical papercrafts, string sculptures, science toys and kits, and many more. It is our belief (and hope) that this landscape of children's crafts can be enriched by modest amounts of computation. A fascinating example in this direction–beads that light up and communicate according to simple procedural rules, and that can be strung together to realize entertaining patterns of light–is described by Resnick *et al.* [15] in their work on "digital manipulatives" for children. The implication of the discussion in this paper is that such digital manipulatives can serve as the means for exploring novel designs for educational programming languages. Again, the scaled-down nature of these languages and the small sizes of digital-manipulative programs, affords the language designer a setting in which a variety of different language models and interfaces can be explored and assessed.

• *Creating Complex Programs.* To date, the sample projects that we have created employ no more than one or two CECIs. This is a natural consequence of the scarcity and unreliability of our prototypes; but we are optimistic enough to believe that eventually, larger constructions with multiple varieties of CECIs working in concert will be achievable. Over time, then, while the individual object programs remain relatively simple, the overall behavior of the larger system of objects could become quite complex. Most likely, the end-user programmers of CECIs will begin to encounter problems such as synchronization and distributed algorithm design–problems that are if anything still thornier than those of debugging and software reuse. Until we have more experience and lore to work with, it will be difficult to predict what sorts of tools might best help users construct such systems; conceivably, applications that simulate multiple real-world programmable objects could help with at least some classes of projects. In any event, our belief is that for many CECI users, the notion of a "computer program" will over time evolve into something less monolithically software-based than it is at present; that is, a "program" will be viewed as a complex mixture of behaviors, some of them taking place in tiny CECI-level computers, some in larger and more powerful computers, and some through the action of real-world materials. The ambitious, longer-term task, then, is for us to reexplore our archetypal notions of software applications and physical objects, as the two realms increasingly diffuse into each other.

ACKNOWLEDGMENTS

We are indebted to the ideas and conversation of Robbie Berg, Ann Eisenberg, Gerhard Fischer, Mark Gross, Mitchel Resnick, and Carol Strohecker, among many others. The work described in this paper has been supported in part by National Science Foundation grants CDA-9616444 and REC-961396, and by a Young Investigator award IRI-9258684 to the first author. The third author is supported by a gift from the Mitsubishi Electric Research Laboratories (MERL) in Cambridge, Massachusetts. Finally, we would like to thank Apple Computer, Inc. for donating the machines with which this research was conducted.

REFERENCES

1. Berlin, A. and Gabriel, K. [1997] Distributed MEMS: New Challenges for Computation.*IEEE Computational Science and Engineering*, Jan-March 1997, pp. 12-16.
2. Blauvelt, G.; Wrensch, T.; and Eisenberg, M. [1999]. Integrating Craft Materials and Computation. To appear in *Proceedings of Creativity and Cognition 3*.
3. Brown, M. [1988] Exploring Algorithms Using Balsa-II, *IEEE Computer*, May, pp. 14-36.
4. Brown, M. and Sedgewick, R. [1985]. Techniques for Algorithm Animation, *IEEE Software,* January, pp. 28-39.
5. Card, S.; Mackinlay, J.; and Robertson, G. [1990]. "The Design Space of Input Devices" *Proceedings of CHI '90*, pp. 117-124.
6. Cypher, A., ed. [1993].*Watch What I Do*. Cambridge, MA: MIT Press.
7. DiGiano, C. [1996]. Self-disclosing design tools: an incremental approach toward end-user programming. Ph.D. thesis, University of Colorado, Boulder.
8. Eisenberg, M. and Eisenberg, Ann N. [1999] Middle Tech: Blurring the Division Between High and Low Tech in Education. In A. Druin, ed. *The Design of Children's Technology*, San Francisco: Morgan Kaufmann, pp. 244-273.
9. Feiner, S.; MacIntyre, B.; and Seligmann, D. [1993] Knowledge-based Augmented Reality. *Communications of the ACM*, 36:7, pp. 52-62.
10. Ghezzi, C. and Jazayeri, M. [1997]. *Programming Language Concepts*, 3rd ed. New York: John Wiley & Sons.
11. Lieberman, H. [1993]. Mondrian: A Teachable Graphical Editor. In A. Cypher (ed.), Watch What I Do. Cambridge, MA: MIT Press.
12. Ada Augusta, Countess of Lovelace. [1842]. Notes on the Analytical Engine. Reprinted in P. and E. Morrison (eds.), *Charles Babbage and his Calculating Engines*, New York: Dover, 1961.
13. Nardi, B. [1993] *A Small Matter of Programming*. Cambridge, MA: MIT Press.
14. Resnick, M.; Martin, F.; Sargent, R.; and Silverman, B. [1996] Programmable Bricks: Toys to Think With. *IBM Systems Journal*, 35:3, pp. 443-452.
15. Resnick, M. *et al.* Digital Manipulatives: New Toys to Think With. *Proceedings of CHI '98*, pp. 281-287.
16. Weiser, M. (1993). Some Computer Science Issues in Ubiquitous Computing. *Communications of the ACM*, 36 (7):75-84.
17. Wrensch, T.; Blauvelt, G.; and Eisenberg, M. [1999]. The Rototack. [*In preparation.*]
18. Wrensch, T. and Eisenberg, M. [1998] "The Programmable Hinge: Toward Computationally Enhanced Crafts" *Proceedings of UIST 98*, San Francisco, November, pp. 89-96.